The Single Vehicle Routing Problem with Deliveries and Selective Pickups in a CPU-GPU Heterogeneous Environment

Igor Machado Coelho, Luiz Satoru Ochi, Pablo Luiz Araújo Munhoz
Institute of Computing
Fluminense Federal University, UFF
Niterói, Brazil
imcoelho@ic.uff.br, satoru@ic.uff.br, pmunhoz@ic.uff.br

Marcone Jamilson Freitas Souza
Department of Computer Science
Federal University of Ouro Preto, UFOP
Ouro Preto, Brazil
marcone@iceb.ufop.br

Ricardo Farias
System Engineering Program-COPPE
Federal University of Rio de Janeiro, UFRJ
Rio de Janeiro, Brazil
rfarias@cos.ufrj.br

Cristiana Bentes
Department of System Engineering
State University of Rio de Janeiro, UERJ
Rio de Janeiro, Brazil
cris@eng.uerj.br

Abstract—In this work, we propose a new algorithm to solve a variant of the Vehicle Routing Problem that is the Single Vehicle Routing Problem with Deliveries and Selective Pickups (SVRPDSP). Our algorithm produces good quality solutions that are better than the best known solutions in the literature. In order to reduce the time spent to solve large-sized instances, we also propose here a parallel implementation of our algorithm that explores a heterogeneous environment composed of a CPU and a GPU. Therefore, our algorithm harnesses the tremendous computing power of the GPU to improve the performance of the local searches computation. We obtained average speedups from 2.73 to 16.23 times with our parallel approach.

Keywords—GPU Computing; Vehicle Routing; Parallel Metaheuristic;

I. INTRODUCTION

The Vehicle Routing Problem (VRP) has attracted a great deal of research attention over the past decades, due to its practical and economical importance in areas like transportation, telecommunication and production planning [1]. Many variants of the VRP have emerged according to the set of constraints applied to the problem. We focus here on the Single VRP with Deliveries and Selective Pickups (SVRPDSP) in which a single vehicle has to deliver goods and collect other goods from the customers. Both pickup and delivery demands are associated with each customer, nonetheless, it is not required that all pickup demands are satisfied. SVRPDSP has relevant applications on logistics operations, where the customers have to send back goods to the depot.

The SVRPDSP is a $NP$-hard problem because it includes the Traveling Salesman Problem (TSP) as a special case when no pickups are required. For this reason, finding the optimal solution to SVRPDSP using exact methods can be computationally infeasible for large-sized problems. Although metaheuristics have proven to provide effective sub-optimal solutions to the VRP, relatively little research has been done on metaheuristics methods to solve the SVRPDSP. In addition, for complex large-scale instances, even metaheuristic methods can spend a considerable amount of time to obtain good quality solutions.

Application of real-world problems are becoming larger and more complex, endorsing the need for high-performance parallel algorithms. Nowadays, Graphics Processing Units (GPUs) offer a tremendous computing power, whose performance has significantly outpaced that of the CPUs. GPUs are massively parallel stream processors with a large number of cores and abundant bandwidth. They are probably today’s most powerful computational hardware per dollar. As a consequence, they have become an attractive high-performance computing platform for general purpose computations, and we are interested in harnessing this power to improve the SVRPDSP response time.

In this paper, we present a new approach to address the SVRPDSP. Our algorithm, called Hybrid General Variable Neighborhood Search (HGVNS), solves the problem using the General Variable Neighborhood Search (GVNS) metaheuristic [2]. GVNS combines a series of random and improving local searches based on systematic changing of neighborhoods. Our results show that the use of this metaheuristic produces solutions that are better than the best known solutions in the literature. For complex large-scale problems, we also propose a parallelization of the GVNS local searches to a heterogeneous environment composed of a CPU and a GPU. The hybrid parallel algorithm uses the strengths of the CPU to choose the neighborhood to be performed, and to search for the best result found. It also takes advantage of the fine-grain parallelism offered by the GPU to perform the Best Improvement local searches.
in the VND method, that is the most expensive part of the computation. Our parallel heterogeneous algorithm also includes memory optimizations and in exploring parallelism between the CPU and the GPU. The parallel algorithm obtained average speedups from 2.73 to 16.23.

The remainder of this paper is organized as following. In the next section, we review the previous work in parallel implementations for the VRP. In section III, we describe our parallel approach. In section IV, we describe our parallel approach. In section V, we report our experimental results. Section VI presents our conclusions and future research plans.

II. RELATED WORK

Since the VRP is a difficult and well-studied combinatorial optimization problem, many different parallel approaches have been developed. A broad summary of the recent parallel approaches for the VRP is given in [3]. In terms of the problem addressed in this paper, SVRPDSP, relatively little research has been conducted in this direction. The difference from the classic VRP is that in SVRPDSP there is a single vehicle (one single route) and it has to deliver goods and optionally return pickup goods back to the depot, if its capacity is not exceeded. If a pickup is made then a monetary profit is gained, so the objective function is to minimize the total traveled distance also subtracting the gains from the pickups.

The SVRPDSP was proposed by [4] and solved by a mixed-integer model with improvement techniques aimed at increasing computational tractability. The work by [5] also proposed a mixed integer linear programming formulation together with classical construction and improvement heuristics, as well as a tabu search heuristic. In [6] it is proposed a new mathematical programming model with a branch-and-cut algorithm for the problem and compared to the exact algorithm presented in [4]. The HGVNS algorithm proposed in this paper was capable of improving the best known solutions for the SVRPDSP presented in [5].

In terms of GPU implementations of metaheuristics, some recent works presented interesting contributions. The GRASP metaheuristic was implemented on the GPU in [7]. Parallel implementations of evolutionary algorithms on GPUs were proposed in [8], [9], [10], and [11]. The works by [12] and [8] presented GPU implementations of the tabu search metaheuristic. Local search on the GPU was also the subject of [13] and [14]. The work by [15] proposed a parallel GVNS algorithm for the GPU. For GVNS, however, the neighborhood structure strongly depends on the target optimization problem. None of these GPU implementations of metaheuristic used the VRP as the target optimization problem.

In terms of the VRP, the work by [16] proposed the implementation of a local search algorithm on the GPU to solve the CVRP, based on [13]. The published work, however, did not show the parallel algorithms or any results. In [17] it is proposed an improved parallel simulated annealing algorithm based on GPU acceleration for solving the VRP with Time Windows. To the best of our knowledge, there is no parallel approach that solves the SVRPDSP in the literature.

III. HYBRID GENERAL VARIABLE NEIGHBORHOOD SEARCH

The algorithm we propose here to solve the SVRPDSP is called Hybrid General Variable Neighborhood Search (HGVNS). It is based on the metaheuristic GVNS, that uses the local search method called Variable Neighborhood Descent (VND). The idea of the algorithm is to escape from local optima trap by changing the neighborhood structure. It explores distant neighborhoods from the current solution, and jumps to a new solution if an improvement has been made.

In HGVNS, the delivery clients are numbered from 1 to n and the pickup clients from n+1 to 2n. Each client i that has demands of both delivery and pickup is duplicated into two clients: a client i for delivery, and a client i+n for pickup. A solution to the SVRPDSP consists of a permutation of the delivery and pickup clients, where the begin and the end are the depot, and all the delivery clients have to show up in the permutation. In the route solution, if a client i precedes a client j, then there is an arc connecting i to j. The evaluation of a solution is based on the costs provided by a cost matrix C, where C(i, j) corresponds to the cost of traveling from client i to client j; a vector P of profits, where P(i) is the profit for a pickup customer i (if i is a delivery customer then P(i) = 0); and a load vector Q, where Q(i) contains the accumulated load of the vehicle in the client i, that cannot exceed the vehicle capacity K for every visited customer.

HGVNS starts by creating an initial solution to the problem. This is done by computing solutions to simpler problems (TSP only with the delivery clients and Knapsack problem only with the pickup clients1), merging these solutions, and performing a local search generating s∗.

The core of HGVNS is to perform local searches on the current solution using the VND method. In VND, different neighborhoods are explored in a specific order, from the fastest to evaluate, to the slower one. The process iterates over each neighborhood while improvements are found. Before doing the local search, the algorithm perform p perturbation steps in order to diversify the solution and effectively escape from local optima. In a perturbation step, the algorithm randomly chooses one neighborhood and applies it on the current solution. After iterMax iterations are performed without improvements, more perturbation steps

1The TSP and Knapsack problems are computed using well-known solvers.
are executed in order to generate more distant solutions. In HGVNS, the following neighborhoods are used as perturbations:

- **Swap** – Permutation between two clients.
- **2-Opt** – Two nonadjacent arcs are removed and another two are added to form a new route.
- **1Or-Opt** – One customer is removed from the current solution and inserted in another position of the solution.
- **2Or-Opt** – Two consecutive customers are removed from the current solution and inserted in another position of the solution.

In order to improve the current solution $s$, in VND all neighbors $s'$ are explored, and since this process has the most expensive computational cost, only the most promising neighborhoods are used (in this order): Swap, 1Or-opt and 2Or-opt, defined as $NS^1$, $NS^2$ and $NS^3$, respectively.

### A. Neighborhood Structures

Consider a problem instance of SVRPDSP that consists of six customers, three for delivery (1 to 3) and three for pickup (4 to 6). Also, consider a valid solution $s$ for the problem without violation of the vehicle capacity where the customers are visited in this order: 2, 3, 6, 1, 4, 5.

After a Swap operation on customers 3 and 1, the customers will be visited in this order: 2, 1, 6, 3, 4, 5.

In case of a 2-Opt operation on the solution $s$, removing arcs (3, 6) and (4, 5), and inserting arcs (3, 4) and (6, 5); the customers will be visited in this order: 2, 3, 4, 1, 6, 5.

When a 1Or-Opt operation is done on the solution $s$ and customer 3 is put after customer 4, the customers will be visited in this order: 2, 6, 1, 4, 3, 5. The process is the similar for 2Or-Opt, but in this case two consecutive customers are moved.

### IV. Heterogeneous Parallel HGVNS

The parallel algorithm we developed is called Heterogeneous Parallel HGVNS (HP-HGVNS). The main idea of the algorithm is to delegate some of the HGVNS work to the CPU and some to the GPU, exploring the different abilities of each processor to deal with specific operations. The GPU is responsible for the most expensive part of the HGVNS, that is to compute the Best Improvement method (BI) for each Neighborhood Structure (NS) into the VND. On the other hand, the CPU is responsible for creating an initial solution, choosing the operation to be performed, and checking for the best solution generated by the GPU.

HP-HGVNS explores the fine-grain parallelism of the GPU in each GPU-BI method. Since it is necessary to apply each neighborhood on all combinations of the $n$ customers, HP-HGVNS creates $O(n^2)$ threads in the GPU. Each HP-HGVNS thread $t_{ij}$ works with the customers $i$ and $j$, in Swap, 1Or-Opt and 2Or-Opt. After applying the move to its pair of clients, the thread must also validate the solution, by checking the load vector $Q$ if it violates the maximum vehicle capacity $K$ at some customer.

The cheapest, among all new solutions created by the GPU-BI method, must be compared to the previous solution, in order to test for improvements. Performing this comparisons on the GPU, however, can be slower than executing this search sequentially on the CPU, since it would require synchronization operations to be carried out simultaneously on the GPU. So, HP-HGVNS takes advantage from the higher performance of the CPU when operating on sequential processing to perform this search. The HP-HGVNS algorithm is depicted in Algorithm 1 and the GPU-BI kernel is shown in lines 15–24 of Algorithm 1.

After the evaluation of the best move, it needs to be applied to the current solution. In HGVNS and HP-HGVNS, the Swap move exchanges customers $i$ and $j$ using an auxiliary variable. In HGVNS, 1Or-Opt inserts customer $i$ after $j$ using one auxiliary variable and a memory copy of a block of customers. The same is done with a 2Or-Opt move, but using two auxiliary variables to put customers $i$ and $i + 1$ after $j$. In HP-HGVNS, the 1Or-Opt and 2Or-Opt moves are applied by means of a specific kernel, where each thread moves one single customer, as in preliminary tests this approach had higher performance than using device-to-device memory copies.

HP-HGVNS starts in the CPU by creating an initial solution to the problem, the same way as done in HGVNS. After that, the CPU copies to the GPU memory the initial solution $s^*$, the cost matrix $C$, and the load vector $Q$. The CPU, then, reserves memory space in the GPU, for the result matrix $R$. With all the structures initialized, the main loop starts. In this loop, the CPU explores the three neighborhoods sequentially, and invokes the GPU kernel to apply the selected neighborhood. The GPU computes the cost of applying the neighborhood in all possible move options, and writes these costs in the result matrix $R$. The CPU copies the result matrix $R$ back, searches for the smallest cost in $R$, and if there is an improvement, apply the equivalent neighborhood move to the solution, and continues the loop, until $p$ reaches $pMax$.

### A. Memory Optimizations

The communication between the CPU and the GPU is done through the GPU global memory. The access to this memory, however, is slow. So, the amount of copies between the CPU and the GPU have to be minimized, and the other levels of the GPU memory hierarchy must be explored. The memory hierarchy is one of the most distinguish features of the current GPUs. It includes the global memory, a local memory for each thread, a shared memory for each block of threads, caches L1 and L2, and two cacheable read-only memories, constant and texture memories. Constant memory is a read-only region of the global memory that speeds up the read requests. Texture memory is another read-only and
Algorithm 1 HP-HGVNS

1: $s_0 \leftarrow \text{Merge (TSP(delivery) + Knapsack(pickup)})$
2: $s^* \leftarrow \text{VND}(s_0)$
3: $p \leftarrow 0$
4: copy $C$ and $s^*$ to the GPU;
5: while $p < p_{\text{Max}}$ do
6: $\text{iter} \leftarrow 0$
7: while $\text{iter} < \text{iter}_{\text{Max}}$ do
8: $s' \leftarrow s^*$
9: for (i = 1 to p + 2) do
10: $k \leftarrow \text{SelectNeighborhood}()$
11: $s' \leftarrow \text{Shake}(s', k)$
12: for (k = 1 to 3) do
13: // VND main loop
14: $Q \leftarrow \text{sum of values for every solution in } s'$
15: // GPU-BI method: call GPU kernel for neighborhood $k$
16: $i \leftarrow i_{\text{_thread}}$
17: $j \leftarrow j_{\text{_thread}}$
18: new cost $\leftarrow \text{cost in } s'$ after NSk move in relation to (i, j)
19: check vector $Q$ for violation of maximum capacity $K$
20: if (vehicle capacity $K$ is exceeded) then
21: $R[i, j] \leftarrow \text{Infinity}$
22: else
23: $R[i, j] = \text{new cost}$
24: // End GPU-BI method
25: copy $R$ from GPU; // Copy result matrix from GPU
26: search for the best solution in $R$
27: $\text{minCost} \leftarrow \text{min}[R[i, j]]$
28: if ($\text{minCost} < \text{cost of } s'$) then
29: $s^* \leftarrow \text{Apply best move of } NS^k \text{ in } s'$
30: $s' \leftarrow s^*$
31: $k \leftarrow 1$
32: if (cost of $s' < \text{cost of } s^*$) then
33: $s^* \leftarrow s'$
34: $p \leftarrow 0$
35: $\text{iter} \leftarrow 0$
36: else
37: $\text{iter} \leftarrow \text{iter} + 1$
38: $p \leftarrow p + 1$

The result matrix $R$, which holds the cost of applying the $NS^k$ move to each pair of clients, is copied to the CPU memory after each kernel execution. To minimize this copy time, we use a pinned-memory in the CPU. This copy allocates memory in the CPU memory space, informing the operating system that the buffer cannot be swapped to disk.

The last issue we investigate is how to efficiently update the solution route $s'$ on the GPU. A naive approach would be to let the CPU modify the route in its memory space, and copy the whole new route to the GPU memory. In HP-HGVNS, instead, we use two structures to store the route in the GPU, the first structure maintains the previous route and the second one maintains the best new route. In the end of each iteration, after the CPU finds the best cost, the new route in the GPU memory must be updated, complying with the winner move. This update is done in the GPU in parallel with the CPU selecting of the next neighborhood to be applied.

V. EXPERIMENTAL RESULTS

In this section we evaluate the performance and the quality of the solutions generated by HP-HGVNS. Our main experimental platform is composed of a CPU Intel i7 2.67 GHz, with 8 GB of memory (only one CPU core was used), and a GPU GeForce GTX 560 Ti, with 1 GB of memory and 384 cores.

HGVNS and HP-HGVNS were implemented in C++, and the GPU part of HP-HGVNS was implemented in CUDA SDK 4.0. The algorithms were tested on 68 problem instances taken from [5].

A. Quality of the Solution

Before analyzing the performance of our heterogeneous parallel algorithm, we have to assess the quality of the solutions produced by the sequential HGVNS algorithm. SVRPDSP is a combinatorial optimization problem, and there is no sense in improving the efficiency of HGVNS, if the algorithm does not produce good solutions to the problem in the first place. The quality of the solution produced by HGVNS depends on the values of the parameters IterMax (maximum number of iterations without improvements), and $p$ (maximum number of perturbations). These parameters were set after some preliminary experiments as $\text{iterMax} = \lfloor N/2 \rfloor$ and $p_{\text{Max}} = \lceil 1.653xN + 95 \rceil$, where $N$ is the number of clients. These parameter values were set identically for all executions. With these settings, we guarantee that there is enough iterations and perturbations to produce good solutions to the problem.

In order to evaluate how good the solutions produced by HGVNS are, we compare them with the lower bounds of the problem. The goal of the SVRPDSP solution is to minimize the cost of the route and to maximize the benefits of the pickups. So, an optimal solution to the TSP provides a lower bound on the cost of the route, while an optimal solution to the Knapsack problem provides an upper bound on the profits of the pickups. The measures of the quality of solution are the same as [5], named cost gap and profit gap. Equation (1) defines the cost gap $c_{\text{gap}}$, that is the gap from the best solution tour size $\text{cost}$ and the minimum tour size $c^*$ of the associated TSP problem. Equation (2) defines the profit gap $p_{\text{gap}}$, that is the gap from the best solution pickup value $\text{profit}$ and the maximum profit $p^*$ of the associated Knapsack problem.

$$c_{\text{gap}} = \left| \frac{c^* - \text{cost}}{\text{cost}} \right| \quad (1)$$
$$p_{\text{gap}} = \left| \frac{\text{profit} - p^*}{p^*} \right| \quad (2)$$

Table I shows the $c_{\text{gap}}$ and $p_{\text{gap}}$ values obtained by HGVNS, compared with the values obtained by the Tabu
Table I
SOLUTION QUALITY RESULTS

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>c_gap(%)</th>
<th>p_gap(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tabu Search</td>
<td>4.03</td>
<td>0.36</td>
</tr>
<tr>
<td>HG-VNS</td>
<td>1.13</td>
<td>0.37</td>
</tr>
</tbody>
</table>

Table II
GPU EXECUTION TIME IN SECONDS FOR THE LOCAL SEARCH METHODS

<table>
<thead>
<tr>
<th>Local Search</th>
<th>C Allocation</th>
<th>Q Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Global</td>
<td>Texture</td>
</tr>
<tr>
<td>Swap</td>
<td>19.54</td>
<td>19.88</td>
</tr>
<tr>
<td>1Or-opt</td>
<td>21.51</td>
<td>22.29</td>
</tr>
<tr>
<td>2Or-opt</td>
<td>24.31</td>
<td>25.28</td>
</tr>
</tbody>
</table>

Search algorithm proposed in [5]. As we can observe in this table, the $c_{\text{gap}}$ results obtained by HG-VNS were much better than the best results obtained in the literature, showing that the algorithm is capable of finding tours with total traveled distance near the lower bound $c^*$. In terms of the metric $p_{\text{gap}}$ there was a difference of only 0.01% between the two algorithms. It is important to notice that for some instances, the solution found by HG-VNS matches the lower bound. These results confirm that HG-VNS obtained good quality solutions for the SVRPDSP. Due to the randomness of the procedure Shake (line 11 of Algorithm 1), the algorithm HG-VNS has to be tested several times. The results reported in this table were obtained by the average of 30 executions for each problem instance.

B. Performance Evaluation

This subsection evaluates the performance of our parallel algorithm. In HP-HGVNS, the most expensive part of the HG-VNS, the VND method, is assigned to the GPU. The VND method consists basically of the different local search application. There are three different local search procedures in HG-VNS, Swap, 1Or-opt and 2Or-opt, put in increasing order of computational cost.

In terms of the memory optimizations implemented in HP-HGVNS, we evaluate different forms of allocating: the cost matrix $C$, the allocation of the load vector $Q$, and the update of the solution route $s_t$. The allocation of $C$ was tested in the global memory and in the texture memory. The allocation of $Q$ was tested in the global memory, and in the shared memory. Table II presents the average execution times (in seconds) for 50 independent runs of each one of HP-HGVNS local search procedures Swap, 1Or-opt and 2Or-opt, considering both forms of allocations for $C$ and $Q$.

The results for the cost matrix $C$ allocation show that using the global memory is more efficient than using the texture memory for all three local search methods. Since the problems are big from the operations research point of view, but still small for parallel algorithms, the overhead of binding/unbinding the data to the texture memory does not surpass the gains of the faster memory. Also, the local search depends on few elements from $C$, that are not located near inside the matrix, so the texture memory cache is not well used. For the allocation of $Q$, using the shared memory is slightly faster than using the global memory, since the complete vector fits in the shared memory and all of its values are needed for the local search. The vector $Q$ is also small compared to common benchmarks for parallel algorithms, so there were little gains by using shared memory, but these gains become more important as the route sizes increase. In terms of the update of the solution route, we compared the timing for the CPU modifying the route with the timing for the GPU modifying the route. The timing results show that using the GPU to update the route is 2 times faster than updating in the CPU. The update of the routes, however, corresponds to only 2% of the GPU execution time. Nevertheless, as the problems size increases, this update will be more significant in the total execution time.

Table III presents the average speedups of the GPU over the CPU for each one of HP-HGVNS local search procedures Swap, 1Or-opt and 2Or-opt, considering the best allocations of $C$ and $Q$, that is, matrix $C$ on global memory and vector $Q$ on shared memory. In this table, we group the problem instances differently from the literature. We create three groups according to the instances sizes: small (16 to 31 clients), medium (33 to 72 clients), and large (76 to 101 clients). The groups small and medium have 28 instances each; and the group large has 12 instances. This new grouping scheme provides more insights into how HP-HGVNS deal with large problems. Comparing the GPU to the CPU, the local search 2Or-opt managed to get a maximum speedup of 43.75 for the bigger instances and performed with speedup 32.88 on average for the complete set of instances. The local search 1Or-opt performed with speedup of 27.26 on average and the Swap local search got speedup values from 1.91 to 14.36. The three proposed local searches managed to improve the CPU times and the Swap had the lowest speedup values and limited the gains of HP-GVNS algorithm. However, to remove the Swap local search from the algorithm impacts negatively on the capacity of finding good quality solutions, so it could not be excluded.

Table IV presents the average execution time (in seconds) of 10 runs of HG-VNS algorithm on the CPU and HP-HGVNS algorithm on the CPU-GPU environment. It also presents the minimum, maximum and average speedups obtained by the parallel execution. The average speedup
values obtained by HP-HGVNS ranged between 2.73 and 16.23, showing the potential of HP-HGVNS to explore the CPU-GPU heterogeneous environment to significantly reduce the execution time of HGVNS.

VI. CONCLUSIONS

In this work, we presented a new algorithm to solve the SVRPDSP, called HGVNS, and its parallelization that explores a heterogeneous environment composed of a CPU and a GPU, called HP-HGVNS. In HP-HGVNS, we explore the strengths of the CPU to choose the neighborhood, and to search for the best result, and the highly parallel architecture of the GPU to perform the most expensive part of the algorithm, that is the local searches. HP-HGVNS also includes some memory optimizations. The way the data structures are allocated in the memory hierarchy of the GPU and the parallelism between the CPU and the GPU influences the execution time.

Analyzing the quality of the results, HGVNS obtained better solutions than the best results obtained in literature. In terms of performance, the 2Or-Opt local search obtained a 43.75 maximum speedup and the 1Or-Opt local search performed with speedup of 27.26 on average. HP-HGVNS explores successfully the GPU parallelism and obtained average speedups from 2.73 to 16.23 for the complete algorithm. The Swap local search had the lowest speedup values, from 1.91 to 14.36, and we intend to improve its efficiency in future works or replace it by another similar local search that keep good quality solutions with better GPU performance.

As future work, we intend to implement HP-HGVNS for a multiple GPUs platform and to extend HGVNS to other Vehicle Routing problems.

ACKNOWLEDGMENT

The authors acknowledge the support from the Brazilian agencies CAPES, CNPq, FAPERJ (grant E-26/110.552/2010) and FAPEMIG.

REFERENCES


